Introduction

1

THESIS

The aggregate of modular syntax definition, generalized scannerless parsing techniques, and parse table composition is the technique for principled combination of multiple textual software languages into a single, composite language.

1.1 LANGUAGE COMPOSITION IN PRACTICE

In modern software development the use of multiple software languages to constitute a single application is ubiquitous. For example, a web application could use *(1)* Java for the main application code *(2)* SQL to query a database, *(3)* HTML to present results in a web browser, *(4)* CSS to style those web pages, *(5)* Javascript to make web pages more interactive, *(6)* XSLT to transform an XML data format to HTML, *(7)* an XML vocabulary to respond to web-service requests, *(8)* XPath to obtain input from web-service requests, *(9)* regular expressions to validate user input, *(10)* an XML configuration format for the application server, *(11)* LDAP search filters for user authentication, *(12)* Unix shell commands to invoke external programs, and *(13)* even the application framework and libraries used by the program constitute a language, though expressed in the syntax and structure of Java. Clearly, the principle of choosing the most suitable software language for the work at hand has deeply penetrated modern software development, resulting in a wide range of *domain-specific languages*. Many of these languages interact closely. For example, the Java web application might compose an LDAP or SQL query based on some user input, pass parameters to an XSLT processor, pass a regular expression to a regular expression library, append a temporary filename to a Unix shell command, and generate some XML fragment. The other way around, the XML configuration file might refer to Java classes and methods to instruct the application server and application frameworks how to respond to incoming requests.

> *However, despite the omnipresent use of combinations of languages, the principles and techniques for using languages together are ad-hoc, unfriendly to programmers, and result in a poor level of integration.*

String Embedding The most recognizable example of a lack of integration is the standard practice of combining a host language (e.g. Java) and some guest languages (e.g. SQL or regular expressions) by writing the guest language programs in string literals of the host program. In this approach, the host language compiler has no understanding whatsoever of the languages that are

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

Figure 1.1 Vulnerable composition of an SQL query, enabling an injection attack

```
SELECT id FROM users
WHERE name = '...'
AND password = '' OR 'x' = 'x'
```

Figure 1.2 Query constructed by the program of Figure 1.1 if the user enters the password `' OR 'x' = 'x`. The where clause is now a tautology.

used in the string literals. This results in a complete lack of static checking of the syntax and semantics of the embedded guest programs, whereas static checking of the host program is often considered to be most valuable. Also, this approach often results in the need for complex escaping of characters with a special meaning in the host language, in particular if combined with related escaping rules of the guest language. Furthermore, using large guest fragments is often awkward if the host language does not support multi-line string literals or *here documents*. Finally, but most importantly, the application might become vulnerable to *injection attacks*, one of the largest classes of security problems, where specially crafted user input results in an unexpected guest program that exposes private information or attacks the functioning of the application.

To prevent injection attacks in the case of dynamically generated sentences, such as SQL queries or shell invocations, the main challenge is to ensure that composing the query with user input is done in a grammatically well-formed way. For example, if the programmer expects the user of the application to provide the content of a literal string in an SQL query, then the user of the application should not be able to craft a query where the user input actually contains other fragments of the guest language, perhaps to construct a tautology (see Figures 1.1 and 1.2). Similar guarantees that composition of guest fragments is done in a grammatically well-formed way are actually already implemented in some metaprogramming systems, but no experiments have been done to bring these techniques to general-purpose programming.

To check the syntax and semantics of the guest language fragments statically, the compiler of the host language needs to be aware of the presence of the guest languages that are used embedded in the program. A guest language plugin to the host compiler could provide the information on how to parse, type-check, and compile the program fragments written in the guest language. To prevent injection attacks, the guest language plugin can provide information to the host compiler on how to construct guest language

```
JTextArea text = new JTextArea(20,40);
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
panel.add(BorderLayout.NORTH, new JLabel("Please enter your message"));
panel.add(BorderLayout.CENTER, new JScrollPane(text));
JPanel south = new JPanel(new BorderLayout(12,12));
JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);
panel.add(BorderLayout.SOUTH, south);
```

Figure 1.3 Sequence of statements for composing a user interface with Swing API methods. The hierarchical structure of the user interface components is not clear.

sentences by composing literal guest language fragments with user input, for example by escaping special characters in the input and checking the lexical structure of the input for characters that are not allowed.

API Syntax A different example of poor integration is the insufficient domain-specific syntax provided by frameworks and libraries. Indeed, most programming languages are designed to allow programmers to implement *semantic* domain abstractions but do not support *syntactic* domain abstraction. Hence, the programmer using the library is forced to use the *generic* lexical syntax and structure of the host language, usually consisting of method invocations, expression that can be nested, and sequences of statements.

> *"An API is like declaring a vocabulary, a domain-specific language adds a grammar which allows to write coherent sentences"* — Martin Fowler

Unfortunately, the generic syntax of a general-purpose language is often not suitable for writing coherent sentences over the vocabulary of an API (see Figure 1.3). To provide a more convenient syntax, the interface to libraries and frameworks is sometimes designed in a way that makes the general-purpose host language look like a domain-specific language (see Figure 1.4). This often involves runtime metaprogramming, heavily exercising the static or dynamic features of the type system of the host language, and frequently results in unusual signatures for functions and methods to provide context to the continuation of the host program. In many cases, methods return a reference to their object (this) only to allow the programmer to write a sequence of method calls. Thus, this approach does not separate the design of the syntactic abstraction from its implementation.

While this approach can lead to remarkable results in programming languages with a generic or liberal syntax, such as Ruby, Haskell, and Lisp, there are always fundamental limitations to the syntax that can be provided to the programmer using the library. Not only are the lexical syntax and the structure of the language limited by the host language, but also the host compiler or interpreter has no understanding of the 'language' defined by the library or framework interface. Hence, it cannot easily provide the programmer with

```
public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));

  auditing.expects(once())
    .method("bought").with(same(TICKET));

  agent.onPriceChange(THRESHOLD);
}
```

Figure 1.4 Testcase implemented using jMock [Freeman & Pryce 2006], where the general-purpose language Java is crafted to look like a domain-specific language. For this test, the mock objects mainframe and auditing are instructed to expect a method invocation

domain-specific error reports about problems with the syntax or semantics of programs. Even languages that are designed to support syntactic domain abstractions, for example using syntax macros, do not allow syntactic extensions without restrictions (Section 2.6.4). Finally, there is usually no separate, formal definition of the syntax of the domain-specific language.

In some cases, very common design patterns and application libraries are lifted to the host language level. For example, LINQ and Cω [Meijer & Schulte 2003a, Meijer & Schulte 2003b] lift the area of data-access to the language C# itself, and the Xtatic language [Xtatic], a follow-up of XDuce [Hosoya & Pierce 2000], extends C# with XML specific support. However, the growth of a language with more domain-specific constructs is limited by the general application area of the language. Therefore, these is a need for the introduction of *concrete syntax* for domain abstractions as a *plugin* to the host language, but without restrictions on the embedding and assimilation of the syntax of the domain-specific language in the host program. The plugin defines *(1)* the concrete syntax for using a specific library, *(2)* in which way the syntax is embedded in the host language, *(3)* how the concrete syntax translates into host language code using the library (a program transformation called *assimilation*), and *(4)* could optionally include typing rules for checking the combination of the host and the guest language.

Effort for Specific Combinations In some cases, specific combinations of languages are crafted carefully because that combination is particularly useful, thus making it acceptable to put more effort in providing a well-engineered combination of these languages to the programmer. We use the term *language conglomerate* for programming languages that are in fact mixtures of various sublanguages. For example, AspectJ has been carefully designed to extend Java with syntax for aspects, advice, and pointcuts; the SQL-92 standard [ISO 1992] defines an embedding of SQL in host languages such as C; and in research the Meta-AspectJ (MAJ) [Zook et al. 2004] code generation tool has integrated the syntax of a particular object language, i.e. AspectJ, in a very user-friendly way in the host language Java.

```
class Foo {
  pointcut foo() : call(void *0.Ef());
}
$ ajc Foo.java
Foo.java:2 [error] Invalid float literal number
pointcut foo() : call(void *0.Ef());
                            ^
```

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
$ ajc Foo.java
Foo.java:2 [error] Syntax error on token "if", invalid allowable
                   token in pointcut or type pattern
```

Figure 1.5 Tokenization bugs in the official AspectJ compiler

However, these languages conglomerates are not a *principled* combination of their sublanguages. That is, the complete syntax of language conglomerates is usually not formally defined, the grammar is not based on separate syntax definitions for the sublanguages, and the applied parsing techniques are ad-hoc, to work around issues in parsing language conglomerates. The use of ad-hoc parsing methods results in compilers that have surprising bugs. Also, the limitations of the parsing techniques influence the design of the languages, which leads to syntax unfriendly to the programmer, e.g. unnecessarily reserving keywords globally. Due to the lack of principled techniques for combining languages, there is a substantial effort for engineering a specific combination of sublanguages into a language conglomerate. Worse, this effort can often not be reused for developing a related conglomerate with a slightly different configuration of sublanguages.

Hence, crafting specific conglomerates does not scale to the full vision of guest languages as plugins to the host compiler. Third parties (i.e. not the developer of the host compiler) should be able to offer plugins and the end-programmer should be able to compose such plugins arbitrarily without doing any difficult metaprogramming or recompilation of the host compiler[1]. In particular, for applications in code generation and query embedding, it is not an option to spend considerable effort to create an implementation for each element of the cross-product of host and guest languages {Java, C#, PHP, Perl, ...} × {SQL, JDOQL, HQL, EJBQL, OQL, LDAP, XML, HTML, XPath, XQuery, Shell, Java, C#, PHP, Perl, ...}. Moreover, it is not acceptable to require implementation effort for every specific *subset* of extensions to the host language. Therefore, while specific combinations are interesting for discovering the requirements for embedded languages, there is a need for guest languages as true plugins to the host language, rather than engineering specific combinations of languages again and again.

[1]Though we want to compose guest languages arbitrarily, some combinations of guest languages inherently do not work together, comparable to incompatible combinations of libraries, e.g. the Swing and SWT user interface toolkits.

1.2 OUTLINE

In this dissertation, we work towards a set of techniques for introducing guest languages in a host language without restrictions on the lexical and context-free syntax of the languages involved. In general, we discuss extensively how the syntax of language conglomerates can formally be defined and parsed. We work towards a *principled* and *generic* solution to language extension by studying the applicability of modular syntax definition, scannerless parsing, generalized parsing algorithms, and program transformations. Where necessary, we extend and improve the existing work in these areas. This dissertation studies a series of compelling state of the art applications of generalized parsing techniques and program transformation, namely

- parsing and disambiguation of syntax embeddings for adding domain-specific syntactic abstractions to a general-purpose language [Bravenboer & Visser 2004 (Chapter 2), Bravenboer et al. 2006],
- concrete object syntax for metaprogramming [Batory et al. 1998, Visser 2002, Bravenboer et al. 2005 (Chapter 3)],
- replacing the fragile and vulnerable practice of embedding languages in string literals [Bravenboer et al. 2007 (Chapter 4)],
- and formal definition of the syntax of language conglomerates, e.g. AspectJ [Bravenboer et al. 2006 (Chapter 5)].

Concerning parsing of language conglomerates, these studies have resulted in considerable more insight in the applications of scannerless parsing and the generalized LR parsing algorithm. Also, the need for programmer-friendly syntax has resulted in novel disambiguation methods for embedded guest languages, relieving end-programmers from the need to indicate the syntactic category of guest language fragments.

The applications we have studied require syntax embeddings to act as true plugins, which can be selected and combined by the end-programmer. The need for a more dynamic configuration of the combination of host and guest languages has resulted in the development of parse table components. Finally, since all this work depends heavily on syntax definitions, solid grammar engineering practices are most important. We improve these practices by introducing a method for recovering and comparing precedence rules of operators from grammars.

Chapters

In Chapter 2 we present the MetaBorg method for introducing concrete syntax for object-oriented libraries and frameworks. This method consists of the embedding of a domain-specific syntax in the general-purpose programming language and an unrestricted program transformation to translate the extended language to the basic host language. The program transformation, called an assimilation, replaces the domain-specific syntax with uses of the

library, which still provides the semantic domain abstraction. This chapter discusses in detail why modular syntax definition and scannerless parsing is important in this application area.

In Chapter 3 we describe an approach to resolving ambiguities that arise when a guest language is embedded in a host language. In particular for metaprogramming, where small program fragments of an object language are quoted, ambiguities are ubiquitous. The approach presented in this chapter makes the existing work on metaprogramming with concrete object syntax [Visser 2002] more programmer-friendly without requiring additional implementation effort for embedding a specific object language. In this chapter, we propose to use a generic extension of the type system of the host language to disambiguate quoted code fragments, thus allowing a more lightweight syntax. As such, it is a generalization of the same programmer-friendly concrete object syntax approach of Meta-AspectJ [Zook et al. 2004].

In Chapter 4 we present the StringBorg method for preventing injection attacks by embedding the syntax of guest languages in a host language. The embedded syntax is translated into invocations of a generated library that guarantees that the sentences are constructed in a grammatically well-formed way. The contribution of this *application* is that it prevents injection attacks *by construction*, rather than *detecting* injection attacks at runtime. To make the syntax of the embedded guest languages as programmer-friendly as possible, we introduce a lightweight disambiguation technique for concrete object syntax. This approach does not depend on the type system of the host language, hence it can be applied to dynamically typed languages, such as PHP. StringBorg is the culmination of genericity: guest language plugins can not only be combined arbitrarily, but can also be used for any host language. For example, a plugin for SQL can be used for PHP as well as Java.

In Chapter 5 we extensively discuss the issues with the parsers of the two main compilers for AspectJ. In particular, we discuss the use of lexical states to tokenize language conglomerates. To improve on the situation that the full syntax of AspectJ is ill-defined, we present a formal definition of the complete syntax of AspectJ. From this formal syntax definition a scannerless generalized LR parser can be generated. To handle the various AspectJ keyword policies, we introduce *grammar mixins*, which are in general useful if the same sublanguage can appear in multiple contexts of a conglomerate.

In Chapter 6 we present an algorithm for composing separately compiled parse table components in a very efficient way just before parsing source files written using a combination of languages corresponding to these components. The parse table composition algorithm allows the syntax of guest language to be deployed as truly separate plugins, which can be combined with the host language and other guest languages efficiently. This solves the problem of having to generate a compound parser from scratch for every particular combination of language extensions. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration (i.e. the set of language extensions) is not fixed.

In Chapter 7 we take a side-path into grammar engineering. All our applications and techniques depend heavily on having high-quality grammars available for the guest and host languages. These grammars have to be defined in the same modular syntax definition formalism, targeting a scannerless generalized LR parser. To make the vision of this dissertation reality, we need solid practices for developing, reverse engineering, and maintaining grammars. In this chapter we solve the particular problem of recovering precedence rules of operators from legacy YACC or SDF grammars and comparing the rules of different grammars to each other. This work enabled the development of a high quality PHP grammar, which we have used for our case studies.

1.3 ORIGIN OF CHAPTERS

Except for our latest work (Chapter 6), all chapters are directly based on peer-reviewed publications at programming languages and software engineering conferences and workshops. Since all chapters are directly based on papers that have been published independently, they can also be read independent of each other. While all papers have distinct core contributions, there is some redundancy in the introduction of background material, motivation, and examples. This redundancy has not been eliminated to make it possible to read the extended and revised papers independently.

- Chapter 2 is a slightly updated version of the OOPSLA 2004 paper *Concrete Syntax for Objects – Domain-Specific Language Embedding and Assimilation without Restrictions.* [Bravenboer & Visser 2004]
- Chapter 3 is an updated and extended version of the GPCE 2005 paper *Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax* [Bravenboer et al. 2005].
- Chapter 4 is an extended version of the GPCE 2007 paper *Preventing Injection Attacks with Syntax Embedding – A Host and Guest Language Independent Approach* [Bravenboer et al. 2007].
- Chapter 5 is a revision of the OOPSLA 2006 paper *Declarative, Formal, and Extensible Syntax Definition for AspectJ – A Case for Scannerless Generalized LR Parsing* [Bravenboer et al. 2006].
- Chapter 6, our latest work, is an unpublished paper *Parse Table Composition – Separate Compilation and Binary Extensibility of Grammars*
- Chapter 7 is an extended version of the LDTA 2007 paper *Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking* [Bouwers et al. 2007].